Formalizing UML/OCL structural features with FoCaLiZe

Messaoud Abbas, Renaud Rioboo, Choukri-Bey Ben-Yelles

Noname manuscript No.
(will be inserted by the editor)

Formalizing UML/OCL structural features with FoCaLiZe

Messaoud Abbas · **Choukri-Bey Ben-Yelles** · **Renaud Rioboo**

Received: date / Accepted: date

Abstract UML (Unified Modelling Language) is the de facto standard for the development of software models and OCL (Object Constraint Language) is used within UML models to specify model constraints. Several UML/OCL tools provide MDE (Model Driven Engineering) transformation into general object oriented programming languages such as Java, C++, etc. But the latter did not provide mechanisms for the specification and the verification of OCL constraints. In this context, formal methods are largely used for the specification of UML/OCL models and the verification of their OCL constraints. However, the divergence between UML (object oriented modelling) and formal methods (mathematical and logical based tools) leads in general to ignore most UML/OCL architectural and conceptual features such as OCL constraints simple and multiple inheritance, late binding, template binding, dependencies, etc. To address the formalization of these features, we have used FoCaLiZe, an object-oriented development environment using a proof-based formal approach. More precisely, we propose a formal transformation of the essential UML/OCL features into FoCaLiZe specifications. The derived formal model reflects perfectly the structural features of the original UML/OCL model. In addition, it is possible to check and prove model properties using Zenon, the automatic theorem prover of FoCaLiZe.

F. Author
College of Exacts Sciences, LABTHOP, El Oued University, 39000 Algeria.

S. Author
Univ. Grenoble Alpes, LCIS, F-26901 Valence, France.

T. Author
ENSIIE, SAMOVAR, Square de la Résistance, F-91025 Evry, France.

Keywords Software Engineering · Model Driven Engineering · Formal Methods · Functional Programming · UML modeling · Semantics.

1 Introduction

Model Driven Engineering (MDE) is a promising approach for software production by automatic models refinements, from abstract specifications to concrete codes. Model Driven Architecture (MDA) Miller et al. (2003) is the OMG[1] particular vision on MDE that relies on the use of UML (Unified Modeling Language) OMG (2015) and OCL (Object Constraint Language) OMG (2014). UML has become a standard to graphically and intuitively describe systems in an object oriented way and OCL allows to enhance UML models with formal constraints. Currently, UML and OCL are largely adopted in software engineering tools to describe structural and behavioral specifications of models.
To ensure the consistency of systems (especially critical ones), it seems relevant to combine UML/OCL with formal methods, that provide mechanisms to express and verify software properties.

As we will see in Section 6, there are numerous works on transforming UML/OCL models using formal tools. Depending on the target language, some important features of object oriented programming are seldom supported. In order to support late binding, multiple inheritance, methods redefinition, dependencies, UML templates and OCL constraint inheritance we decided to use the FoCaLiZe environment Hardin et al. (2016), a formal language inspired from Coq Coq (2016) and OCaml[2] functional paradigms with additional object oriented features. So, the paper presents formal transformations of UML/OCL models into FoCaLiZe

[1] OMG: Object management group http://www.omg.org/
[2] Objective Caml programming language: http://ocaml.org/

specifications that maintain the aforementioned UML/OCL features.

Both FoCaLiZe developers and UML developers can benefit from such transformations. From the UML point of view, FoCaLiZe could be exploited as a model verification tool. On the other hand, from the FoCaLiZe point of view, it is possible to use a UML/OCL model as a starting point for a FoCaLiZe development.

The reminder of this document is structured as follows: First, Sections 2 and 3 present the essential UML/OCL supported features and the main concepts of FoCaLiZe. Then, Section 4 details the correspondences between UML/OCL and FoCaLiZe. After that, Section 5 shows an overview on the usefulness of the proposed transformation. Before concluding, Section 6 presents comparison with related works on the transformation of UML features into formal methods.

2 UML/OCL

UML is a general-purpose modeling language that helps developers to design, visualize and document the artifacts of software engineering. A UML model is a set of diagrams describing the static and the behavioral aspects of software systems. OCL is a declarative language for describing constraints on UML elements.

The current study supports a subset of UML2 class diagram features and a subset of the OCL constraints. The main ensured features are:

- UML classes with attributes and operations,
- inheritance and multiple inheritance with method redefinition and late binding,
- multiple dependency relationships,
- UML templates and template bindings,
- OCL invarants, pre-conditions and post-conditions,
- and OCL constraints (invariant, pre and post-conditions) inheritance.

OCL constraints inheritance means that the OCL constraints of super-classes are automatically transmitted through simple and multiple inheritances to their sub classes.

The UML/OCL syntax follows the official UML and OCL **metamodels** OMG (2015, 2014). However, we prefer to represent (during the description of the transformation rules) the UML/OCL elements in EBNF notation (rather than standard XML notation), in order to increase the readability of our transformation rules.

The next paragraphs detail the UML/OCL supported syntax. We describe first the UML class diagram constructs, then we present the supported OCL expressions.

2.1 UML Class Diagram

A class diagram is a set of UML classes with possibly UML relationships. The general definition of a UML class (named) *cn* is expressed as follows:

$$[\textbf{public}][\ll class-stereotype\gg]\ \textbf{class}\ cn\ (\mathbb{P}_{cn})$$
$$\textbf{binds}\ \mathbb{T}_{cn}\ \textbf{depends}\ \mathbb{D}_{cn}\ \textbf{inherits}\ \mathbb{H}_{cn}\ =\ \mathbb{A}_{cn}\ \mathbb{O}_{cn}\ \textbf{end} \quad (1)$$

with

- $\mathbb{P}_{cn}$ a list of formal parameters declarations,
- $\mathbb{T}_{cn}$ a list of substitutions of formal parameters with actual parameters,
- $\mathbb{H}_{cn}$ a list of class names from which the current class inherits,
- $\mathbb{D}_{cn}$ a list of class names to which the class *cn* depends,
- $\mathbb{A}_{cn}$ a list of attributes and
- $\mathbb{O}_{cn}$ a list of operation.

We shall first describe the main members of a UML class: attributes and operations, then we detail relationships between classes.

The list of attributes $\mathbb{A}_{cn}$ $(attr_1 \ldots attr_k)$ characterizes the state of the class *cn* instances. The general syntax of attributes definition is:

$$\textbf{attribute}\ attrVis\ attrNane : typeExp\ [mult] \quad (2)$$

The non-terminal symbol *mult* specifies the multiplicity of the attribute. If *mult* is different from 1 (the default value), the attribute is multivalued. In this paper, we focus on the general cases of attributes visibility: `+` (`public`) and `-` (`private`).

The list of operations $\mathbb{O}_{cn}$ $(op_1 \ldots op_k)$ manipulates the attributes of the class *cn* and changes the status of its instances. Each operation has form:

$$\textbf{operation}\ V\ S\ N \begin{pmatrix} dir_1\ p_1 : type_1\ [mult_1], \\ \ldots, \\ dir_m\ p_m : type_m\ [mult_m] \end{pmatrix} : Type\ [mult] \quad (3)$$

Each operation has a name *N* and specified by a visibility *V* and a stereotype *S*. In this paper, we consider the general cases of operation visibilities : `+` (`public`) or `-` (`private`). The stereotype *S* may be **create** for class constructor. The operation parameter directions dir_i are either **in** (by default) or **out**. The p_i are the operation parameter names and $type_i$ their types. Operation parameter multiplicities ($mult_i$) are similar to attributes multiplicities. The operation return type is *Type* and has multiplicity *mult*.

The inheritance is the mechanism that enables a new subclass *cn* to acquire all attributes and operations of its superclasses $(cn_1 \ldots cn_k)$. The class *cn* can also be enriched with its own attributes and operations, as it can redefine (override) operations of the super-classes:

$$\textbf{public class}\ \ cn\ \textbf{inherits}\ cn_1,\ \ldots,\ cn_k\ =\ \ldots\ \textbf{end} \quad (4)$$

A UML template class cn is a general class that is parameterized by a list of formal parameter names $(fp_1 \ldots fp_k)$. Each formal parameter is specified by a type expression:

public class $cn\ (fp_1 : type_1, \ldots, fp_k : type_k) = \ \ldots$ **end** (5)

Template classes cannot be directly instantiated, they should be refined using template binding relationships to create concrete bound models. A bound model cn' can be derived from a template class cn through substitutions of its formal parameters in a dedicated binding relationship. The latter specifies a list of template parameter substitutions ($\mathbb{T}_{cn}$) that associates actual elements (of the bound model) to formal parameters (of the template):

public class cn' **bind** cn

$$<fp_1 \texttt{ -> } ap_1, \ \ldots, fp_k \texttt{ -> } ap_k> = \ \ldots \textbf{ end} \quad (6)$$

where fp_i `->` ap_i indicates the substitution of the formal parameter fp_i with the actual parameter ap_i. Standard constraints require the type of each actual parameter (in the bound model) to be a sub-type of the corresponding formal parameter. In the particular case of formal template parameters of the type **Class**, they can be substituted by any class of the model. The type **Class** is a super-type of all classes.

A UML dependency is a relationship which indicates that the specification of a class cn (client) requires supplier classes $(cn_1, \ldots, cn_k)$. This implies that the definition of the client class can use the supplier classes in order to develop its own methods. Contrary to the inheritance mechanism, the client class does not acquire or redefine the attributes and operations of the supplier classes, it can only use them. We use the clause **depends** to specify a dependency:

public class cn **depends** $cn_1, \ \ldots, cn_k \ = \ \ldots$ **end** (7)

2.2 OCL Constraints

An OCL constraint is a statement of the OCL language OMG (2014) which uses types and operations on types. We distinguish between primitive types (`Integer`, `Boolean`, `Real` and `String`), enumeration types, object types (classes of UML model) and collection types. For a given type `T`, the OCL type `Collection(T)` represents a collection family of elements of type `T`.

OCL constraints are OCL invariants, pre-conditions and post-conditions. Each OCL constraint is expressed by a formula following the OCL **metamodel** syntax OMG (2014).

An invariant is an OCL constraint attached to a class that must be true for all instances of that class at any time. Given a class cn, an OCL invariant ($\mathbb{E}_{inv}$) associated to the class cn, has the following form:

`context` cn `inv :` $\mathbb{E}_{inv}$ (8)

where $\mathbb{E}_{inv}$ is the logical expression (formula) describing the invariant.

The pre-condition describes a constraint that should be true before the operation is executed and the post-condition describes a constraint which must be satisfied after the operation is executed. Given a class cn, an OCL pre-condition ($\mathbb{E}_{pre}$) and post-condition ($\mathbb{E}_{post}$) associated to an operation of the class cn has general form:

`context` cn `::`$N(p_1 : type_1 \ldots p_k : type_k)$

pre :$\mathbb{E}_{pre}$ **post :**$\mathbb{E}_{post}$ (9)

where N is the operation name, $p_1 \ldots p_k$ are the operation parameters and $type_1 \ldots type_k$ their corresponding types. $\mathbb{E}_{pre}$ and $\mathbb{E}_{post}$ are logical expressions (formulas) describing the pre/post-conditions.

2.3 Example of a UML/OCL Model

In this paper, we have chosen a UML/OCL pedagogic and theoretical example in order to consider most UML/OCL architectural and conceptual features such as UML templates, bound models, OCL constraints inheritance, . . . which are rarely gathered in the same project. Otherwise, several real projects have been developed using FoCaLiZe and UML/OCL such as the verification of the UML/OCL model of a control system of a railway crossing a road Abbas et al. (2018) and the verification of the UML/OCL model of the airport security regulations Delahaye et al. (2008b). Additional applications are also available in the web site of the transformation tool: `http://www.univ-eloued.dz/uml2foc/`.

One of the usual application of UML templates and template bindings is the modeling of generic classes (as C++ templates), that enables developers to avoid repetition and enhance the re-usability of codes. We present here an example (see Fig. 1) of a UML template modeling a generic class (`FArray`) and its specialization (`FStack` and `PersonStack`) using inheritance and binding relationships. We remind that the formal parameter `T:Class` of the template `FArray` could be substituted by any class of the model, to generate new bound models.

The annotations attached to the classes `FArray`, `FStack` and `Person` are OCL constraints following the OCL **metamodel** syntax OMG (2014).

The first invariant, of the class `FArray` specifies for each array `a`, whether it is empty then it is not full. The second invariant species that the length of each empty array (`a`) equals to zero (`a.length()=0`).

The OCL constraints of the class `FStack` are pre/post-conditions. The first pre/post-conditions (of the operation `push(t:T)`) specifies that when stacking an element in an empty stack and then unstacking it out, the stack remains

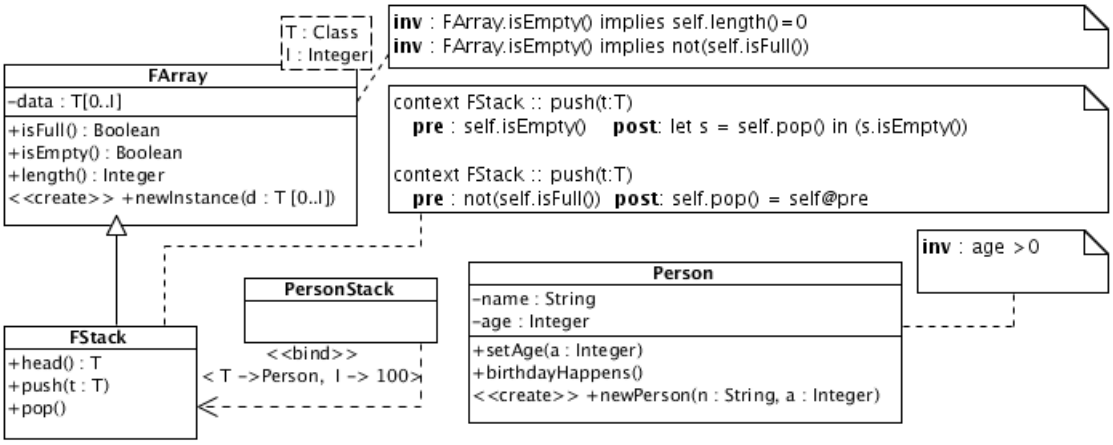


Fig. 1: Example of UML/OCL Model

empty. The second pre/post-conditions (of the same operation) specifies that when stacking an element in a not full stack and then unstacking it out, the stack remains unchanged.

Finally, the invariant of the class `Person` specifies that the value of the its attribute `age` is always greater than 0.

2.4 Dependency Graph

Given a UML class diagram, before performing its transformation into FoCaLiZe, we start by ordering the classes according to the syntactic and semantic correlations between them.

A UML class cn_i is syntactically and semantically dependent on another class cn_j in the following cases:

- The class cn_i inherits from the class cn_j ($cn_j \in \mathbb{H}_{cn_i}$).
- The class cn_i is parameterized by the class cn_j ($cn_j \in \mathbb{P}_{cn_i}$).
- The class cn_i is dependent on the class cn_j ($cn_j \in \mathbb{D}_{cn_i}$).
- The class cn_i is a bound class obtained through a binding relationship from the class cn_j ($cn_j \in \mathbb{T}_{cn_i}$).

In such way, we construct a **dependency graph** that will guide the transformation of the UML class diagram into FoCaLiZe.

3 FoCaLiZe

FoCaLiZe is a complete development environment that integrates a **programming** language, a **requirement** specification language and a **proof** language. A FoCaLiZe development is built step by step starting from abstract specifications until reaching concrete implementations using object oriented features Ayrault et al. (2009); Delahaye et al. (2008a); Fechter (2005). To verify and analyze software properties, FoCaLiZe is based on Coq.

The main brick is a **species**, a modular structure that groups together the carrier type of the species, functions to manipulate this carrier type and logical properties.

Table 1: The Syntax of a Species

spec	::=	**species** *species_name* [(*param* [{ , *param*}*])] **=** [**inherit** *spec_def* [{ , *spec_def* }*] ;] {*methods*;}* **end**;;
param	::=	*ident* **in** *type* | *ident* **is** *spec_def*
spec_def	::=	*species_name* | *species_name* (*param* [{ , *param*}*])
methods	::=	*rep* | *signature* | *let* | *property* | *theorem*
rep	::=	**representation** **=** *type*;
signature	::=	**signature** *function_name* **:** *function_type*;
let	::=	**let** [**rec**] *function_name* **=** *function_body*;
property	::=	**property** *property_name* **:** *property_specification* ;
theorem	::=	**theorem** *property_name* **:** *property_specification* **proof** **=** *theorem_proof*;

The general syntax of a species (see Table 1) uses the following methods:

- The **representation** to describe the data structure of the species instances,
- **signatures** specifying functions without giving their computational bodies (only the functional type is provided in the species),
- **let** definitions to specify functions together with their computational bodies,
- **property** statements to express properties (requirements) that should be satisfied in the context of the species (no proof is provided in the species),

- **theorem** statements to express properties together with their formal proofs.

A species can be built from scratch or using (multiple) inheritance from existing species. It can also be prametrized by existing species, to express dependency relationships between species. So, using inheritance and parameterization, a FoCaLiZe development is designed as a hierarchy of species. Note that, FoCaLiZe supports methods redefinition, except for representation, and late binding mechanisms.

To illustrate the above FoCaLiZe concepts, we present a concrete development that aims to model points (of the plan) and circles. Each circle is defined by its radius and its center (a point):

Code 1: **The species Point and Circle**

```
 (* Definition of the type color *)
type color = | Red | Green | Blue ;;

species Setoid =
 signature equal: Self-> Self-> bool;
 signature element: Self;
 property equal_reflexive: all x: Self,
                           equal (x, x);
 property equal_symmetric: all x y: Self,
              equal(x, y) -> equal(y, x);
 property equal_transitive:all x y z:Self,
  equal(x, y)-> equal(y, z)-> equal(x, z);
 end;;
species Point = inherit Setoid;
 signature getX : Self -> float;
 signature getY : Self -> float;
 signature move :
           Self -> float -> float -> Self;
 (* distance: calculates the distance
               between two given points *)
 let distance (a:Self, b: Self):float =
   sqrt( ((getX(a) - getX(b))*
         (getX(a) - getX(b))) +
         ((getY(a) - getY(b))*
         (getY(a) - getY(b)))) ;
 (* distanceSpecification: specifies
                    the method distance *)
 property distanceSpecification:
     all p q:Self, equal(p, q) ->
                 distance(p, q) = 0.0;
 end;;

species Circle (P is Point) =
 representation = P * float ;
 let newCircle(centre:P, radius:float):
                 Self = (centre, radius);
 let getCenter(c:Self):P = fst(c);
 let getRadius(c:Self):float = snd(c);
 end;;
species ColoredPoint = inherit Point;
 representation =  (float * float) * color;
 let getColor(p:Self):color = snd(p);
 let newColoredPoint( x:float,
                      y:float,
                      c:color):Self=
                      ((x, y), c);
 let getX(p) = fst(fst(p));
 let getY(p) = snd(fst(p));
 let move(p, dx, dy) =
      newColoredPoint( getX(p) + dx,
                       getY(p) + dy,
                       getColor(p)   );
 let element =
          newColoredPoint(0.0, 0.0, Blue);

 proof of distanceSpecification =
                by definition of distance;
 proof of   equal_reflexive = assumed;
 proof of   equal_symmetric = assumed;
 proof of   equal_transitive = assumed;
 end;;
```

- At top level (outside the species), we define the sum type `color` that will be used by the species of the example.
- The general structure `Setoid` models any non-empty sets with an equivalence relation. Because the representation is still undefined, it is possible to inherit from this species to construct new species with different representations.
- Although the species `Point` (modeling points of the plan) inherits signatures and properties of `Setoid`, its representation is still undefined.
- The species `Circle` is parameterized by the species `Point` in order to define the center of a circle. Here, the species `Circle` can use all signatures, functions and properties of `Point`, even if they are not completely defined yet.
- The species `ColoredPoint` (specifies colored points) is a complete species that inherits the species `Point` and provides definitions for all its signatures and properties (including inherited signatures and properties).

FoCaLiZe provides several means to write the proofs of properties. We can directly write Coq proofs or using the key word **assumed** to accept proofs without providing proofs. But the usual way to write proofs consists to use the FoCaLiZe proof language (FPL). Using FPL, the developer organizes the proof in steps. Each step provides proof hints that will be exploited by Zenon (the automatic theorem prover of FoCaLiZe) Bonichon et al. (2007).

4 From UML/OCL to FoCaLiZe

After the description of UML/OCL elements, we present now the transformation process of a UML/OCL model into FoCaLiZe:

- To start with, we deal with the transformation of a UML class without relationships with other classes.
- Then, we present the mapping of OCL constraints, class invariants and pre/post-conditions.
- After that, we describe the transformation of relationships between classes: multiple inheritance, dependency, templates, template bindings and associations.

- Finally, we present the general transformation of a UML class diagram, UML classes with possibly relationships and OCL constraints.

The transformation we have defined uses similarities between UML/OCL concepts and FoCaLiZe concepts Delahaye et al. (2008a); Fechter (2005). A UML class corresponds to a FoCaLiZe species, class attributes correspond to the species representation, class operations correspond to species signatures and OCL constraints correspond to species properties. Also, UML and FoCaLiZe share similar relationships and mechanisms such as (multiple) inheritance, dependency, parameterization, methods overriding and late binding.

4.1 Transformation of classes without relationships

FoCaLiZe species have a larger abstraction level than UML classes. In fact, a FoCaLiZe species corresponds to a UML meta class. In particular, each class element has a direct counterpart in a FoCaLiZe species. This has led us to transform a UML class into a FoCaLiZe species, class attributes to species signatures (modeling attributes getters) and class operations (which are only declared) to species signatures.

Before dealing with the transformation of attributes and operations, we present first the transformation of UML type expressions, including UML primitive types and UML enumeration types.

Notations and conventions
Throughout the next sections, we will use the following notations and conventions:

- For a UML/OCL element e, we denote $[\![e]\!]$ its transformation into FoCaLiZe.
- For a UML/OCL element named *en*, we maintain the same name for its transformation into FoCaLiZe, taking into account upper and lower cases to respect FoCaLiZe syntax.

All UML primitive types have their FoCaLiZe counterparts:

Primitive Types:

$$\begin{array}{ll} [\![\texttt{Integer}]\!] & = \texttt{int} \\ [\![\texttt{Real}]\!] & = \texttt{float} \\ [\![\texttt{Boolean}]\!] & = \texttt{bool} \\ [\![\texttt{String}]\!] & = \texttt{string} \\ [\![\texttt{UnlimitedNatural}]\!] & = \texttt{unlimited\_Nat} \end{array}$$

A general UML type expression is followed by an integer interval ($typeExp$ `[`$mult$`]`) specifying its multiplicity. It is converted into a FoCaLiZe type as follows:

$$[\![\ typeExp\ \texttt{[}mult\texttt{]}\ ]\!] = \begin{cases} [\![typeExp]\!] & \text{if } mult = \texttt{1..1} \\ list([\![typeExp]\!]) & \text{if } mult \neq \texttt{1..1} \end{cases} \quad (10)$$

Attributes (see formula (2)) specify the states of class objects. Then, each attribute gives rise to a signature modeling its getter function in the corresponding species:

UML:
$attrNane$ `:` $typeExp$ `[`$mult$`];`
FoCaLiZe:
`signature` $get\_attrNane$ `: Self ->` $[\![\ typeExp\ \texttt{[}mult\texttt{]}\ ]\!]$`;`

Operations (see formula (3)) represent services invoked by any object of the class in order to affect object behaviors. In the context of object oriented programming languages, when an instance o of the class c_n invokes an operation named N with the stereotype S of the class, the memory state of the instance o is affected and moves to a new memory state o'. In functional languages (without memory state) such as FoCaLiZe, the two memory states of an object represent two different entities. Taking into account this difference between the two formalisms, we convert a class operation into a species signatures (function interface) that starts with the type `Self` (the entity that invokes the function), followed by the function parameter types and ends with the type `Self` (the new created entity). So, the general transformation of operations (see formula (3)) is:

UML:

$$S\ N \left(\begin{array}{l} dir_1\ p_1 : type_1\ [mult_1] \\ \ldots \\ dir_k\ p_k : type_k\ [mult_k] \end{array} \right) : Type[mult]$$

FoCaLiZe:

$$\begin{array}{l} \texttt{signature } N\texttt{: [Self]->}[\![Type_1[mult_1]]\!]\texttt{->...->} \\ \qquad\qquad\qquad\qquad [\![Type_k[mult_k]]\!]\texttt{->[Self];} \end{array}$$

Note that if the operation is a class constructor ($op\_st =$ `<<create>>`), its transformation will not start with the type `Self`. Also, if the class operation returns a particular value (has a returned type, $returnType$), its transformation will end with the FoCaLiZe type [[returnType[mult]]].

Using the transformation rules for attributes and operations, a UML class cn (without relationships with other classes) specified with the attributes list $\mathbb{A}_{cn}$ and the operations list $\mathbb{O}_{cn}$ is then converted into a FoCaLiZe species (named) sn, with signatures $[\![\mathbb{A}_{cn}]\!]$ and $[\![\mathbb{O}_{cn}]\!]$, as follows:

UML:
[**public** |**private** |**protected**] [**final** |**abstract**]
[«class-stereotype»] **class** $cn = \mathbb{A}_{cn}\quad \mathbb{O}_{cn}$
end
FoCaLiZe:
`species` $sn = [\![\mathbb{A}_{cn}]\!]\ [\![\mathbb{O}_{cn}]\!]$
`end ;;`

4.2 Mapping of OCL Constraints

To transform OCL expressions we had to built a FoCaLiZe library that formalizes OCL expressions. In this library,

Classes of the UML model correspond to FoCaLiZe species and OCL primitive types correspond to FoCaLiZe primitive types.

The OCL constraints (invariants, pre-conditions and post-conditions) specified in the context of a UML class are then mapped into FoCaLiZe properties of the corresponding species.

To OCL primitive types (`Integer`, `Real`, `String`) correspond equivalent FoCaLiZe primitive types (as we did for UML primitive types). Most of the OCL operations on integers are then directly converted into their corresponding FoCaLiZe operations using the same notations:

Table 2: Mapping of OCL **Integer** expressions

OCL	FoCaLize
n	n
$\alpha+\beta$	$[\![\alpha]\!]$ + $[\![\beta]\!]$
$\alpha-\beta$	$[\![\alpha]\!]$ - $[\![\beta]\!]$
$-\alpha$	-$[\![\alpha]\!]$
$\alpha*\beta$	$[\![\alpha]\!]$ * $[\![\beta]\!]$
α **div** β	$[\![\alpha]\!]$ / $[\![\beta]\!]$
α **mod** β	$[\![\alpha]\!]$ % $[\![\beta]\!]$
α.**min**(β)	`min`($[\![\alpha]\!]$, $[\![\beta]\!]$)
α.**max**(β)	`max`($[\![\alpha]\!]$, $[\![\beta]\!]$)
α.**abs**	`abs`($[\![\alpha]\!]$)

The transformation of `String` and `Real` expressions are handled in a similar way, using FoCaLiZe operations on `string` and `float` types.

Most OCL formulas (of type `Boolean`) have a straightforward counterpart as FoCaLiZe boolean expressions:

Table 3: Mapping of OCL formulas

OCL	FoCaLiZe
true	**true**
false	**false**
not(ϕ)	~~($[\![\phi]\!]$)
ϕ **and** ψ	$[\![\phi]\!]$ /\ $[\![\psi]\!]$ / $[\![\phi]\!]$ && $[\![\psi]\!]$
ϕ **or** ψ	$[\![\phi]\!]$ \/ $[\![\psi]\!]$ / $[\![\phi]\!]$ || $[\![\psi]\!]$
ϕ **xor** ψ	$[\![\phi]\!]$ |<>| $[\![\psi]\!]$
ϕ **implies** ψ	$[\![\phi]\!]$ -> $[\![\psi]\!]$
if ϕ **then** ψ **else** φ	**if** $[\![\phi]\!]$ **then** $[\![\psi]\!]$ **else** $[\![\varphi]\!]$
let $x : type = Exp$ **in** ϕ	**let** $x = [\![Exp]\!]$ **in** $[\![\phi]\!]$
$\alpha = \beta$	$[\![\alpha]\!] = [\![\beta]\!]$
$\alpha <> \beta$	~~($[\![\alpha]\!] = [\![\beta]\!]$)
$\alpha > \beta$	$[\![\alpha]\!] > [\![\beta]\!]$
$\alpha < \beta$	$[\![\alpha]\!] < [\![\beta]\!]$
$\alpha >= \beta$	$[\![\alpha]\!] >= [\![\beta]\!]$
$\alpha <= \beta$	$[\![\alpha]\!] <= [\![\beta]\!]$
allInstances -> forAll($x \mid \phi$)	**all** x : `Self`, $[\![\phi]\!]$
allInstances -> exists($x \mid \phi$)	**ex** x : `Self`, $[\![\phi]\!]$

Where ϕ and ψ are two OCL formulas, α and β are two integer/real expressions.

Note that, the OCL operations `forAll` and `exists`, when applied to the OCL collection returned by the `allInstances` operation, are respectively mapped to the FoCaLiZe universal (`all`) and existential (`ex`) quantifiers. Otherwise, they must be turned into special defined FoCaLiZe operations that iterate on all instances of a collection.

An OCL invariant (see formula (8)) $\mathbb{E}_{inv}$ of the class *cn* is converted into a FoCaLiZe property of the corresponding species:

OCL:
`context cn inv :` $\mathbb{E}_{inv}$
FoCaLiZe:
`property` *invIdent* `: all` *e* `: Self,` $[\![\mathbb{E}_{inv}]\!]$ `;`

The OCL formula $\mathbb{E}_{inv}$ is converted into the FoCaLiZe boolean expression $[\![\mathbb{E}_{inv}]\!]$ using the correspondences between OCL and FoCaLiZe presented in Tables 3 and 2.

An OCL pre and post-conditions $\mathbb{E}_{pre}$ and $\mathbb{E}_{post}$ of an operation named *N* of the class *cn* (see formula (9)) are transformed together into a FoCaLiZe implication (*pre-condition* $\Rightarrow$ *post-condition*) of the corresponding species:

OCL:
`context` *cn* `::` $N(p_1 : type_1 \ldots p_k : type_k)$
pre :$\mathbb{E}_{pre}$ **post** :$\mathbb{E}_{post}$
FoCaLiZe:
`property` *prePostIdent* `:`
`all` *e* `: Self,`
`all` $p_1 : [\![type_1]\!]$ `, ... , all` $p_k : [\![type_k]\!]$ `,`
$[\![\mathbb{E}_{pre}]\!]$ `->` $[\![\mathbb{E}_{post}]\!]$ `;`

As for invariants, the OCL formulas $\mathbb{E}_{pre}$ and $\mathbb{E}_{post}$ are converted into FoCaLiZe boolean expressions ($[\![\mathbb{E}_{pre}]\!]$ and $[\![\mathbb{E}_{post}]\!]$) using the correspondences between OCL and FoCaLiZe presented in Tables 3 and 2. More details about the correspondence between OCL and FoCaLiZe expressions are given in Abbas (2014).

The transformation of the class `Person` (see Fig. 1) illustrates the above transformation rules of classes, attributes, operations and OCL constraints as follows (Code 2):

Code 2: **Transformation of the class Person**

```
species Person =
  signature get_name : Self -> string;
  signature get_age : Self -> int;
  signature setAge : Self -> int -> Self;
  signature birthdayHappens: Self -> Self;
  signature newPerson : string -> int
                                -> Self;
  property inv_1: all p:Self,
                    (get_age(p) > 0);
  end;;
```

The signatures `get_age` and `get_name` show the transformation of the class `Person` attributes (`name` and `age`). The signature `setAge` corresponds to the operation having the same name of the class `Person` and the signature `transforms` the class `Person` constructor (preceded by the stereotype `<<create>>`, see Fig. 1). Contrary to UML object

oriented paradigm, the entities (of the species Person) that will invoke these signatures are explicit (the first `Self` of each signature), except for the species constructor. This is due to the functional paradigm of FoCaLiZe language.

The property `inv_1` corresponds to the class `Person` invariant (`age > 0`).

4.3 Transformation of Relationships Between Classes

We present now the transformation of the supported UML relationships, which are:

- inheritance and multiple inheritance with method redefinition and late binding mechanisms,
- multiple dependency relationships,
- UML templates and template binding relationships.

To transform (multiple) inheritance, templates, template bindings and dependency features, we use similar mechanisms in FoCaLiZe. Although FoCaLiZe is a functional language, it supports multiple inheritance, parameterized species (like templates in UML) and the substitution of formal parameters of species, which is similar to template binding in UML.

A UML template *cn* (see formula (5)) with formal parameters $fp_1 : type_1, \ldots, fp_k : type_k$ corresponds to a parameterized species, where each formal parameter of the template is transformed into a species parameter:

UML:
public class $cn\ (fp_1 : type_1, \ldots, fp_k : type_k) = \ldots$ **end**
FoCaLiZe:
`species` $sn(fp_1$ `is|in` $[\![type_1]\!], \ldots,$
fp_k `is|in` $[\![type_k]\!])$`= ... end;;`

Formal template parameters of type `Class` are modeled as formal species parameters of `Setoid` (using the keyword `is`). Formal template parameters of primitive types are modeled as formal species parameters of FoCaLiZe primitive types (using the keyword `in`). More details about UML templates transformation are provided in Abbas et al. (2014).

The template class `FArray` (see Fig. 1) has two parameters: `t` of type `Class` and `i` of type `Integer`. It is translated into a parameterized species (having the same name) with two formal parameters as follows:

Code 3: **Transformation of the class FArray and its OCL constraints**

```
 (* Transformation of the class FArray *)
species FArray ( Obj is Setoid,
                 i   in IntCollection) =
 signature get_data : Self -> list(Obj);
 signature isFull : Self -> bool ;
 signature isEmpty : Self -> bool ;
 signature length: Self -> int ;
 signature newInstance: list(Obj) -> Self;

 (* mapping of OCL constraints on FArray*)
 property inv_1 : all s : Self,
           isEmpty(s) -> (length(s) = 0);
 property inv_2 : all s : Self,
            isEmpty(s) -> ~~(isFull(s));
 end;;
```

The class formal parameter of the template `FArray (T: Class)` is transformed into the species formal parameter `Obj is Setoid`. The primitive formal parameter of the template `FArray (i: Integer)` is transformed into the species formal parameter `i in IntCollection`. In fact, we have used the FoCaLiZe corresponding collection (`IntCollection`) to create an integer entity.

The UML (multiple) inheritance mechanism between the subclass *cn* and the super classes $cn_1, \ldots, cn_k$ (see formula (4)) is directly transformed into (multiple) inheritance between their corresponding species, using the clause **inherit** of FoCaLiZe:

UML:
public class cn **inherits** $cn_1, \ldots, cn_k = \ldots$ **end**
FoCaLiZe:
`species` $sn(\mathbb{P}_{sn})$ `= inherit` $sn_1(\mathbb{P}_{sn_1}), \ldots sn_k(\mathbb{P}_{sn_k})$`; ... end;;`

Where, $\mathbb{P}_{sn}$ is the parameters list of the derived species *sn* and $\mathbb{P}_{sn_i}$, $i : 1..k$ are the parameters list of the derived species sn_i, $i : 1..k$. Note that in the particular case when the classes $cn_1, \ldots, cn_k$ are not parameterized, the transformation becomes:

`species` sn `= inherit` $sn_1, \ldots, sn_k$`; ... end;;`

All OCL constraints of the class *cn* (including those inherited from the classes $cn_1 \ldots cn_k$) become properties of the the species *sn* through multiple inheritance.

Code 4 shows the transformation of the inheritance relationship between the classes `FArray` and `FStack` (see Fig. 1). This relationship is transformed into FoCaLiZe inheritance between the species `FArray` (derived from the class `FArray`) and the species `FStack` (derived from the class `FStack`):

Code 4: **Transformation of the class FStack**

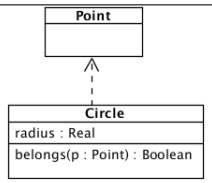
```
 (* Transformation of the class FStack *)
species FStack ( Obj is Setoid,
                 i in IntCollection ) =
                   inherit FArray(Obj, i);
 signature head : Self -> Obj ;
 signature push : Obj -> Self -> Self;
 signature pop  :  Self -> Self;

 (* mapping of OCL constraints on FStack*)
 property pre_post_push_1: all e: Obj,
                           all s: Self,
  isEmpty (s) -> isEmpty(pop(push(e, s)));

 property pre_post_push_2: all e:Obj,
                           all s:Self,
  ~(isFull(s))-> equal(pop(push(e, s)), s);
 end;;
```

Table 4: Transformation of a dependency relationship

UML	FoCaLiZe

	`species Point = ... end;;`
	`species Circle (P is Point)=`
	`signature get_radius: Self -> float;`
	`signature belongs: Self -> P -> bool;`
	`...`
	`end;;`

The dependency between the client class cn and the supplier classes $cn_1, \ldots, cn_k$ (see formula (7)) enable us to define the attributes, operations and OCL constraints of cn using those of the classes $cn_1, \ldots, cn_k$. Hence, we transform a dependency relationship between classes using FoCaLiZe parameterization:

UML:
public class cn **depends** $cn_1, \ldots, cn_k = \ldots$ **end**
FoCaLiZe:
`species` $sn($ $\mathbb{P}_{sn_1}, \ldots, \mathbb{P}_{sn_k}$, fp_1 `is` $sn_1(\mathbb{P}_{sn_1}), \ldots, fp_k$ `is` $sn_k(\mathbb{P}_{sn_k}))$ `=` `...` `end;;`

Where, $\mathbb{P}_{sn_i}$, $i : 1..k$ are the parameters list of the species sn_i, $i : 1..k$ (derived from the classes $cn_1, \ldots, cn_k$). Note that in the particular case when the classes cn, $cn_1, \ldots, cn_k$ are not parameterized, the transformation becomes:

`species` $sn(fp_1$ `is` $sn_1, \ldots, fp_k$ `is` $sn_k)$ `=` `...end;;`

Like in UML, The properties of supplier species (derived from OCL constraints of supplier classes) can be safely used through the parameterization by the client species, to develop its own methods.

To decide whether a given point `p` belongs to a given circle `c`, the class `Circle` (see Table 4) uses the operation *belongs* $(p : Point)$ parameterized with an instance of the class `Point`. This expresses a dependency relationship between the classes `Circle` (the client) and `Point` (the supplier).

To transform a binding relationship (see formula (6)), we use both inheritance and parameterization mechanisms in FoCaLiZe. Actual parameters are provided using FoCaLiZe collections and parameter substitution mechanisms. For the binding of primitive formal parameters, we directly use the FoCaLiZe corresponding collections (`IntCollection`, `FloatCollection`, ...) to create primitive entities. A formal parameter P of type `Integer` is bounded in FoCaLiZe by the creation of integer entities using `IntCollection` as follows:

UML:
p `->` *integerValue*
FoCaLiZe:
`let p = IntCollection!createInt(` $[\![integerValue]\!]$ `);;`

Formal parameters on other primitive types are bounded in a similar way.

The substitution (binding) of the formal parameter `T` of type `Class` with an effective parameter (`T->cn'`) is transformed into a FoCaLiZe substitution (binding), as follows:

UML:
`T ->` cn'
FoCaLiZe:
`T is` $[\![$`cn'`$]\!]$

Code 5 shows the transformation of the binding relationship between the template class `FStack` and the class `PersonStack` (see Fig. 1):

Code 5: **Transformation of the class PersonStack**

```
(* creation of the entity medeling
                  the integer value 100 *)
let e = IntCollection!newInstance(100);;
(* Transformation of the
                      class PersonStack *)
species PersonStack( T is Person,
                     i in IntCollection)=
          inherit FStack(T, e);
end;;
```

On the UML side, the OCL constraints of the template `FStack` are propagated to its bound model (`PersonStack`). This mechanism is also maintained on the FoCaLiZe side, the properties of the species `FStack` (derived from the OCL invariant and pre/post-conditiond) are propagated to the species `PersonStack`.

4.4 Transformation of Classes with Relationships

Given a class cn with attributes, operations and having relationships (multiple inheritances, dependencies and template bindings) with other classes, we can now generate its corresponding species using the following general transformation:

UML:
[**public**] [«class-stereotype»] **class** cn $(\mathbb{P}_{cn})$
binds $\mathbb{T}_{cn}$ **depends** $\mathbb{D}_{cn}$ **inherits** $\mathbb{H}_{cn}$= $\mathbb{A}_{cn}$ $\mathbb{O}_{cn}$
end
FoCaLiZe:
`species` sn $(\mathbb{P}_{sn})$ `=`
`inherit Setoid,` $[\![\mathbb{H}_{cn}]\!]$ `;` $[\![\mathbb{A}_{cn}]\!]$ $[\![\mathbb{O}_{cn}]\!]$ `end ;;`

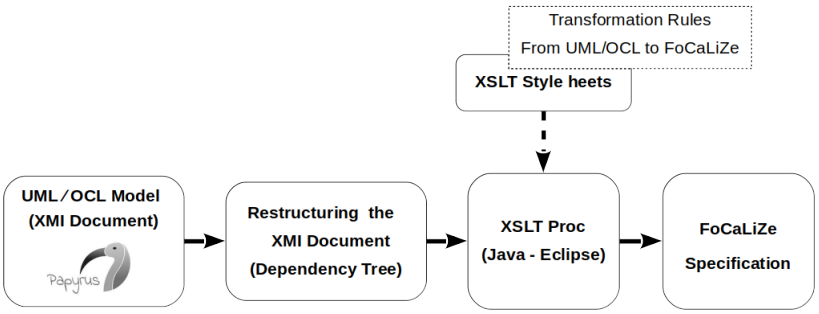


Fig. 2: Systematic transformation of UML models into FoCaLiZe

Where, $\mathbb{P}_{sn}$ is the list of parameters of the species *sn*, provided from the transformation of the class *cn* parameters ($[\![\mathbb{P}_{cn}]\!]$), parameters substitutions ($[\![\mathbb{T}_{cn}]\!]$) and dependencies ($[\![\mathbb{D}_{cn}]\!]$). The list of species from which the species *sn* inherits is obtained from the transformation of the class *cn* inheritances list ($[\![\mathbb{H}_{cn}]\!]$). Finally, the transformation of the class attributes list ($[\![\mathbb{A}_{cn}]\!]$) and the transformation of the class operations list ($[\![\mathbb{O}_{cn}]\!]$) generate the signatures of the species *sn*.

The complete generated FoCaLiZe specification from the UML/OCL model of Fig. 1 is presented in appendix A.

4.5 Deployment

The proposed transformation from UML/OCL to FoCaLiZe is implemented using the XSL Transformations (XSLT) W3C (2014). Recommended by the World Wide Web Consortium (W3C), the XSLT is a usable language for the transformation of XML documents into various formats such as HTML, XML, text, PDF, etc. We have developed XSLT stylesheets specifying the transformation rules from a UML model expressed in the XMI interchange format (generated by the **Papyrus** graphical tool) into FoCaLiZe. The output is the corresponding FoCaLiZe source file, which can be directly read by the FoCaLiZe compiler (see Fig. 2). Additional information about the achieved transformation tool (**UML2FOC**) are now available on-site (`http://www.univ-eloued.dz/uml2foc/`), where instructions for installation and use are detailed.

4.6 Correctness of the transformation rules

In order to ensure the correctness of our transformation rules we have defined the following semantics (for both UML and FoCaLiZe):

- Γ_U for UML
- Γ_F for FoCaLiZe.

For a given class named *cn*, $\Gamma_U(cn) = (cn, V_{cn})$, where V_{cn} is the value of the class. A class value is a pair $(\Gamma_{cn}, body_{cn})$ composed of the local context of the class (Γ_{cn}) and the body of the class ($body_{cn}$). The body of the class is composed of class attributes ($\mathbb{A}_{cn}$), class operations ($\mathbb{O}_{cn}$) and class constraints ($\mathbb{C}_{cn}$). In other words, the body of the class *cn* is expressed as follows (the trailing star sign $*$ denotes several occurrences):
$body_{cn} = (\mathbb{A}_{cn}, \mathbb{O}_{cn}, \mathbb{C}_{cn})$, where

- $\mathbb{A}_{cn} = \{(attr\_n : typeExp)\}^*$, where $attr\_n$ is an attribute name of the class *cn* and $typeExp$ its type.
- $\mathbb{O}_{cn} = \{(op\_n : opType)\}^*$ where op_n is an operation name of the class *cn* and $opType$ its parameters and returned types.
- $\mathbb{C}_{cn} = \{(const\_n : invExp)\}^*$ where $const\_n$ is a constraint name of the class *cn* and $invExp$ its first order expression.

The local context of the class *cn* (Γ_{cn}) is the list of the class *cn* parameters ($\mathbb{P}_{cn}$) and the list of the class *cn* dependencies ($\mathbb{D}_{cn}$), see class definition formula (1):
$\Gamma_{cn} = \mathbb{P}_{cn} \bigcup \mathbb{D}_{cn}$ where,
$\mathbb{P}_{cn} = \{(fp\_n,\ typeExp)\ /\ fp\_n\ \in \mathbb{P}_{cn}\}$ and
$\mathbb{D}_{cn} = \{cn_i\ /\ cn_i \in \mathbb{D}_{cn}\}$

Using the above definitions, we define **by induction**, the general semantics Γ_U of a UML/OCL model $\mathbb{M}$ with multiple inheritance relationships (see class definition, formula (1)), formal parameters, dependencies list and OCL constraints, as follows:

- $\Gamma_U = \{(cn,\ (\Gamma_{cn},\quad \mathbb{A}_{cn},\ \mathbb{O}_{cn},\ \mathbb{C}_{cn}))\ /\ cn\ \in \mathbb{M}\}$, where
- $\Gamma_{cn} = \mathbb{P}_{cn} \bigcup \mathbb{D}_{cn}$
- $\mathbb{A}_{cn} = \mathbb{A}_{cn} \bigcup_i \{\mathbb{A}_{cn_i}\ /\ cn_i \in \mathbb{H}_{cn}\}$, $\mathbb{H}_{cn}$ is the list of inherited classes.
- $\mathbb{O}_{cn} = \mathbb{O}_{cn} \bigcup_i \{\mathbb{O}_{cn_i}\ /\ cn_i \in \mathbb{H}_{cn}\}$, $\mathbb{H}_{cn}$ is the list of inherited classes.
- $\mathbb{C}_{cn} = \mathbb{C}_{cn} \bigcup_i \{\mathbb{C}_{cn_i}\ /\ cn_i \in \mathbb{H}_{cn}\}$, $\mathbb{H}_{cn}$ is the list of inherited classes.

For our concern, the derived FoCaLiZe model will be composed of a list of abstract species (they only contain signatures and declared properties). This is due to the fact that original UML/OCL models are also abstract, and contain no implementations. So, each derived species sn is specified by:

- A list of species names from which the current species inherits ($\mathbb{H}_{sn}$),
- A list of signatures (sig_{sn}) and
- A list of properties ($prop_{sn}$).

In a symmetrical way, for a given species named sn, $\Gamma_F(sn) = (sn, V_{sn})$, where V_{sn} is the value of the species. A species value is a pair $(\Gamma_{sn}, body_{sn})$ composed of the local context of the species (Γ_{sn}) and its body ($body_{sn}$). The body of the species sn is denoted: $body_{sn} = (Sig_{sn}, Prop_{sn})$, with

- $Sig_{sn} = \{(funName : funSig)\}^*$ where $funName$ is a function name and $funSig$ its signature.
- $Prop_{sn} = \{(propName : propSpec)\}^*$ where $propName$ shows the name of a property and $propSpec$ its logical statement.

So, we define the general semantics Γ_F of a FoCaLiZe model $\mathbb{F}$ with a multiple inheritances list $\mathbb{H}$, formal parameters list $\mathbb{P}$ and properties specification as follows:

- $\Gamma_F = \{(sn, (\Gamma_{sn}, sig_{sn}, prop_{sn})) / sn \in \mathbb{F}\}$, where
- $\Gamma_{sn} = \mathbb{H}_{sn} \bigcup \mathbb{P}_{sn}$
- $\mathbb{P}_{sn} = \mathbb{P}_{sn} \bigcup_{i=1}^{k} \{\mathbb{P}_{sn_i}\} \bigcup_{i=1}^{k} \{(fpn_i \texttt{ is } sn_i(\mathbb{P}_{sn_i}))\}$
- $sig_{sn} = sig_{sn} \bigcup_{i} \{sig_{sn_i} / sn_i \in \mathbb{H}_{sn}\}$, $\mathbb{H}_{sn}$ is the list of inherited species.
- $prop_{sn} = prop_{sn} \bigcup_{i} \{prop_{sn_i} / sn_i \in \mathbb{H}_{sn}\}$, $\mathbb{H}_{sn}$ is the list of inherited species.

During the transformation of the class cn to the species sn, both $\Gamma_U(cn)$ and $\Gamma_F(sn)$ are enriched progressively. In such a way, we define a one-to-one correspondence where each new UML definition of the set $\Gamma_U(cn)$ is paired with exactly one element of $\Gamma_F(sn)$, and each element of $\Gamma_F(sn)$ is paired with exactly one element of $\Gamma_U(cn)$. Using the aforementioned reliable semantic and the proposed transformation rules (from UML/OCL to FoCaLiZe), we prove that the properties of a UML model are maintained in its FoCaLiZe transformation.

5 Usefulness of the transformation

As we mentioned in the introduction of this document, the proposed transformation is meant to be useful for both FoCaLiZe and UML users:

- **For UML Users**, FoCaLiZe can analyze and check the properties of of UML/OCL models. In fact, FoCaLiZe is motivated by its powerful automated theorem prover Zenon Doligez (2016) and its proof checker Coq Coq (2016). Realizing proofs with Zenon makes the user intervention much easier since it manages to fulfill most of the proof obligations automatically. In addition, whenever such a proof fails, Zenon helps the user to locate the source of the inconsistency.
For example (see Code 6), we can check the correctness of the property `pre_post_push_1` (derived from the first pre/post condition of the operation `pus(t:T)`) using the following FoCaLiZe proof hints :

Code 6: **Proof of the property pre_post_push_1**

```
species PersonStack ( T is Person,
                      i in IntCollection)=
                      inherit FStack(T, e) ;
  ...
 (* Proof of the property
                       pre_post_push_1 *)
  proof of pre_post_push_1  =
 <1>1 assume e : T, s : Self,
      hypothesis H1: isEmpty(s),
      prove isEmpty(pop(push(e, s)))
 <2>1 prove equal(pop (push(e , s) ), s)
      by hypothesis H1  property inv_2,
      pre_post_push_2
 <2>2 prove isEmpty(pop (push (e, s)))
     by hypothesis H1 step  <2>1
     property inv_1, equal_symmetric
 <2>3 qed by step <2>2
 <1>2 conclude ;
  ...
 end;;
```

In fact, we try to prove one property (`pre_post_push_1`) using the other properties (`inv_2`, `pre_post_push_2`, ...) as premises. This mechanism helps developers to detect eventual contradictions between the properties of the same UML/OCL model.
Let us note that if a proof fails, the FoCaLiZe compiler indicates the line of code responsible for the error. There are two main kinds of errors: either Zenon could not find a proof automatically, or there are inconsistencies in the original UML/OCL model. In the first case, the developer interaction is needed to give appropriate hints to prove the properties, while in the second case one must go back to the original UML/OCL model to correct and/or complete it.

- **For FoCaLiZe Users**, it is possible to use a UML/OCL model as the starting point for a FoCaLiZe development. Having provided the transformation rules from UML/OCL to FoCaLiZe, we can now generate a FoCaLiZe abstract specification, starting from a UML class diagram annotated with OCL constraints. Then, FoCaLiZe developers will be able to complete the generated specification, by implementing all its undefined methods

(representations + signatures), until reaching the final code.

Refinements of the leaf species generated from the UML/OCL model of Fig. 1 is detailed in appendix B.

We note that whatever the definitions provided for the generated signatures, they must satisfy the properties derived from OCL constrains. Otherwise, FoCaLiZe will prevent developers from creating collections for such species.

6 Related Works

Various approaches have been used to provide supporting tools and techniques to formalize UML models and check OCL constraints. The most widely used are set theory based methods such as the B method Abrial (2005) and Alloy Jackson (2012). Several approaches use first and higher-order logic based tools (Maude Clavel et al. (2007) and Isabelle/HOL Nipkow et al. (2002)). Other approaches and techniques use Real time checker tools (such as PROMELA, SMV and LOTOS). Moreover, there are numerous works focusing on specific aspects of UML/OCL using formal methods such as rCOS (A refinement calculus of object systems) Ke et al. (2012) and CASL (Common Algebraic Specification Language) Mosses (2004).

B is a method for specifying, designing and coding software systems based on set theory. Among works interested on the transformation of UML/OCL models into the B method based tools, some focus on the study of the transformation rules and verification techniques Tao and Fengsheng (2015); Truong and J. (2006), while others concentrate on the generation of concrete transformation tools (ArgoUML+ B Ledang et al. (2003), UML2B Snook and Butler (2004) and UML-B Snook and Butler (2006); Said et al. (2015)).

Most of UML features and OCL constraints are supported in the transformation into B method.

However, the multiple inheritance mechanism, UML templates (with template bindings) and the propagation of OCL constraints are not supported.

Alloy is a formal language with restricted set-based syntax, for describing structural properties. It has been recently used to formalize and check the consistency of UML/OCL models Cunha et al. (2013); Nimiya et al. (2010); Cunha et al. (2015). Using Alloy, a UML class is modeled by a signature (a set of atoms). Simple inheritance is considered through the clause `extends`, and some kind of parameterization is taken care of at module level. However, as for B, the multiple inheritance cannot be directly represented in Alloy Anastasakis et al. (2010). Furthermore, UML templates and template bindings are not supported, and it is not possible to import abstract modules (at specification level) and bind modules parameters to create actual modules.

Several approaches based on equational logic, first-order logic and higher-order logic are also used through different formal tools.

ITP-OCL tool Clavel and Egea (2006) is a rewriting-based tool that supports automatic validation of UML class diagrams with respect to OCL constraints.

KeY Beckert et al. (2007) proposes a mapping from OCL into first-order logic that allows interactive reasoning about UML diagrams with OCL constraints.

HOL-OCL Brucker and Wolff (2007); Rull et al. (2015) maps OCL constraints into higher-order logic. It concentrates on producing OCL evaluator tools. HOL-OCL supports most of UML/OCL features, in particular simple inheritance (expressed as inclusion of sets) and late binding mechanisms.

The Maude system is based on rewriting logic and runtime checking tools. It is also used as target language in the formalization of UML/OCL models Chama et al. (2015); Durán et al. (2011). The simple inheritance feature is supported through the definition of predicates which establishes that one class is a subclass of another class. Moreover, Maude allows to specify a module parameterized with formal parameters. We think that this feature enables to formalize UML template and template binding.

But, although HOL-OCL and transformation to Maude are the most comprehensive tools, they also ignore multiple inheritance, UML templates, template bindings and the propagation of OCL constraints.

Some other approaches interested in UML models verification use real time checker: SMV Kwon (2000), PROMELA (using Spin model checker) Lilius and Paltor (1999) and LOTOS (using CADP model checker) Carreira and Costa (2003). They concentrate on the transformation of UML state machines. But, they ignore OCL constraints and are limited to the verification of isolated UML state machines.

Other formal tools and techniques are concerned with specific aspects of UML/OCL models and the study of their semantics. In particular, UMLtoCSP Cabot et al. (2007) provides a mapping of OCL constraints into constraint programming expressions to check UML class diagrams annotated with OCL constraints. Some other proposals have used CASL Reggio et al. (2001); Favre (2010) in order to provide formal semantics for UML diagrams. Finally, a formalization of UML models using rCOS is proposed Yang (2009) in order to study the consistency conditions among a number of related UML models.

Likewise, essential UML/OCL features as multiple inheritance and template bindings are not taken care of.

Comparison with Related Works

The perfect formal modeling of UML/OCL models depends on the used formal method. Three essential aspects charac-

Table 5: Comparison - From UML/OCL to Formal methods

		Transformation to B Method	Transformation to Alloy	Transformation to Maude	Transformation to HOL (HOL-OCL)	Transformation to FoCaLiZe
UML/OCL Architectural and Conceptual Features Supported by the Transformation	Encapsulation	•	•	•	•	•
	Simple Inheritance	•	•	•	•	•
	Dependency	•	•	•	•	•
	Multiple Inheritance					•
	Methods redefinition				•	•
	Late binding				•	•
	Templates specification	•	•	•	•	•
	Template Binding specification			•		•
	OCL constraints	•	•	•	•	•
	OCL constraints inheritance					•
	Propagation of OCL constraints through dependency and template binding					•
Verification techniques	Model checker and/or Test Generation		•	•		•
	Theorem Prover	•		•	•	•
Programming Paradigm	Imperative	•		•		
	Functional				•	•

terize formal methods: the supported architectural and conceptual object-oriented features, the paradigm and the expressiveness of its programming language and the available verification and proof techniques.

As matter of comparison, we have studied the transformation of the UML/OCL model of Fig. 1 into FoCaLiZe and into similar formal development environments. The class `PersonStack` is created through inheritance and template binding from the classes `FArray`, `FStack` and `Person`. Thus, it acquires all methods and OCL constraints of these classes. Using our transformation approach, we get a formal and abstract model (see appendix A) that reflects perfectly the original UML/OCL model with all its methods, properties, relationships and conceptual (specification) scope. All these features are maintained within a purely functional paradigm scope and with the possibility to check properties using either the theorem prover of FoCaLiZe or test generation techniques. Contrary to transformations to the aforementioned formal environments (B method, Alloy, Maude and HOL-OCL), it is not possible to preserve the properties of the original UML/OCL model. This is because the based formal methods do not allow features like inheritance, multiple inheritance, methods redefinition, late binding, OCL constraints, OCL constraints inheritance and template bindings. That leads in general to generate a concrete and manufactured model rather than abstract and conceptual one. Table 5 recapitulates the result of comparison between the most powerful transformation tools. It highlights the capacity of supporting high architectural and conceptual features, verification techniques and functional programming paradigm. This comparison ensures that the transformation to FoCaLiZe maintains the highest standards in software development.

7 Conclusion and Perspectives

In this paper we have been able to produce a formal transformation from UML/OCL into FoCaLiZe that reflects most of UML class diagram functionality such as multiple inheritance, dependency, templates, template bindings and the navigation of OCL constrains over these features.

Starting from a UML class diagram annotated with OCL constraints, we generate a FoCaLiZe specification, where each class corresponds to a species, class attributes are modeled as getter functions, class operations are converted into signatures of the corresponding species and OCL constraints are mapped into species properties.

More precisely, the proposed transformation naturally preserves the following UML/OCL aspects:

- **The (multiple) inheritance with methods overriding and late binding**: We obtain a species hierarchy that perfectly reflects the original UML inheritance relationships between classes.
- **UML templates, template bindings and dependencies using species parameterization and parameters substitution**: A species derived from one class can be used as a parameter of another species derived from another class, even at the specification level. The FoCaLiZe collection and late-binding mechanisms ensure that all methods appearing in a species (used as formal parameter) are indeed implemented and all properties are proved.

- **All species properties derived from OCL constraints are propagated through FoCaLiZe inheritance and parameterization relationships**: The properties of a super-species become properties of its inheritors (sub-species), the properties of a supplier species can be safely used by client species and the properties of a species derived from a UML template become properties of the species derived from the corresponding bound models.

A direct application of the proposed approach consists in combining UML/OCL and FoCaLiZe in MDE modellingmeworks to generate certified applications.

In parallel with the presented approach, we are working on the transformation of UML diagrams describing the behavioral aspect of UML classes. We have first dealt with state machine diagrams Abbas et al. (2018), and now we are working on the transformation of UML activity diagrams. The idea consists in modeling a UML activity diagram by the definition of a recursive function using the pattern matching mechanism.

Finally, we note that (by construction) the proposed transformation from UML/OCL to FoCaLiZe is not bidirectional. However, we plan to study more in details the composition of our proposal and the inverse transformation from FoCaLiZe to UML Delahaye et al. (2008a). We believe indeed that the two formalisms could further cooperate, using the graphical expressiveness of UML/OCL and the FoCaLiZe proof capabilities.

8 Compliance with Ethical Standards

Conflict of interest: The authors declare that they have no conflict of interest.

Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

References

Abbas M (2014) Using FoCaLiZe to check OCL constraints on UML classes. In: International Conference on Information Technology for Organization Development, IT4OD 2014, Conference proceedings, pp 31–38

Abbas M, Ben-Yelles CB, Rioboo R (2014) Modeling UML Template Classes with FoCaLiZe. In: The International Conference on Integrated Formal Methods, IFM2014, Springer, LNCS, vol 8739, pp 87–102

Abbas M, Ben-Yelles CB, Rioboo R (2018) Modelling UML state machines with FoCaLiZe. International Journal of Information and Communication Technology 13(1):34–54

Abrial JR (2005) The B-Book: Assigning Programs to Meanings. Cambridge University Press

Anastasakis K, Bordbar B, Georg G, Ray I (2010) On challenges of model transformation from UML to Alloy. Software & Systems Modeling 9(1):69–86

Ayrault P, Hardin T, Pessaux F (2009) Development life-cycle of critical software under FoCaL. Electronic Notes in Theoretical Computer Science 243:15–31

Beckert B, Hähnle R, Schmitt P (2007) Verification of object-oriented software: The key approach. LNAI, 4334, Springer-Verlag pp 1–18

Bonichon R, Delahaye D, Doligez D (2007) Zenon: An extensible automated theorem prover producing checkable proofs. In: Logic for Programming, Artificial Intelligence, and Reasoning, Springer, pp 151–165

Brucker A, Wolff B (2007) The HOL-OCL Book. Swiss Federal Institute of Technolgy (ETH), available at: http://www.brucker.ch/

Cabot J, Clarisó R, Riera D (2007) Umltocsp: a tool for the formal verification of UML/OCL models using constraint programming. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, pp 547–548

Carreira PJ, Costa ME (2003) Automatically verifying an object-oriented specification of the steam-boiler system. Science of Computer Programming 46(3):197–217

Chama W, Chaoui A, Rehab S (2015) Formal Modeling and Analysis of Object Oriented Systems using Triple Graph Grammars. International Journal of Embedded and Real-Time Communication Systems (IJERTCS) 6(2):48–64

Clavel M, Egea M (2006) ITP/OCL: A rewriting-based validation tool for UML+ OCL static class diagrams. In: 11th Int. Conf. on Algebraic Methodology and Software Technology, LNCS 4019, Springer, pp 368–373

Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2007) All about Maude a High Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer-Verlag

Coq (2016) The Coq Proof Assistant, Tutorial and Reference Manual, Version 8.5. INRIA – LIP – LRI – LIX – PPS, Distribution available at: http://coq.inria.fr/

Cunha A, Garis A, Riesco D (2013) Translating between Alloy Specifications and UML Class Diagrams Annotated with OCL Constraints. Software & Systems Modeling 14(1):5–25

Cunha A, Garis A, Riesco D (2015) Translating between Alloy specifications and UML class diagrams annotated with OCL. Software & Systems Modeling 14(1):5–25

Delahaye D, Étienne J, Donzeau-Gouge V (2008a) Producing UML Models from Focal Specifications: An Application to Airport Security Regulations. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, pp pp. 121–124

Delahaye D, Étienne JF, Donzeau-Gouge VV (2008b) A formal and sound transformation from Focal to UML: an application to airport security regulations. Innovations in Systems and Software Engineering 4(3):267–274

Doligez D (2016) The Zenon Tool. Software and Documentations freely available at http://focalinriafr/zenon/

Durán F, Gogolla M, Roldán M (2011) Tracing properties of UML and OCL models with Maude. arXiv preprint arXiv:11070068 [csSE]

Favre LM (2010) Formalization of mof-based metamodels. Model Driven Architecture for Reverse Engineering Technologies Information Resources Management Association pp 49–79

Fechter S (2005) Sémantique des traits orientés objet de FoCaL. PhD thesis, Université PARIS 6

Hardin T, Francois P, Pierre W, Damien D (2016) FoCaLiZe : Tutorial and Reference Manual, version 0.9.1. CNAM-INRIA-LIP6, available at: http://focalize.inria.fr

Jackson D (2012) Software Abstractions: logic, language, and analysis. MIT press, Cambridge, Great Britain

Ke W, Li X, Liu Z, Stolz V (2012) rCOS: A formal model-driven engineering method for component-based software. Frontiers of Computer Science 6(1):17–39

Kwon G (2000) Rewrite rules and operational semantics for model checking UML statecharts. In: UML2000 - The Unified Modeling Language, Springer, LNCS, vol 1939, pp 528–540

Ledang H, Souquières J, Charles S, et al. (2003) Argouml+ B: Un Outil de Transformation Systématique de Spécifications UML en B. In: AFADL'2003, pp 3–18

Lilius J, Paltor IP (1999) vUML: A tool for verifying UML models. In: Automated Software Engineering, 14th IEEE International Conference on., pp 255–258

Miller J, Mukerji J, et al. (2003) MDA guide. Object Management Group, Inc, version 101

Mosses P (2004) CASL reference manual: The complete documentation of the common algebraic specification language, vol 2960. LNCS, Springer

Nimiya A, Yokigawa T, Miyazaki H, Amasaki S, Sato Y, Hayase M (2010) Model checking consistency of UML diagrams using Alloy. World Academy of Science, Engineering and Technology 71(99):547–550

Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: A Proof Assistant for Higher-order Logic, vol 2283. LNCS, Springer

OMG (2014) OCL : Object constraint language 2.4 Available at: http://www.omg.org/spec/OCL/2.4

OMG (2015) UML : Unified Modeling Language, version 2.5. Available at:http://www.omg.org/spec/UML/2.5/PDF

Reggio G, Cerioli M, Astesiano E (2001) Towards a rigorous semantics of UML supporting its multiview approach. In: Fundamental Approaches to Software Engineering, LNCS 2029, pp. 171–186, 2001, Springer-Verlag, Berlin Heidelberg 2001

Rull G, Farré C, Queralt A, Teniente E, Urpí T (2015) AuRUS: explaining the validation of UML/OCL conceptual schemas. Software & Systems Modeling 14(2):953–980

Said MY, Butler M, Snook C (2015) A method of refinement in UML-B. Software & Systems Modeling 14(4):1557–1580

Snook C, Butler M (2004) U2B: A tool for translating UML-B models into B. in J Mermet (ed), UML-B Specification for Proven Embedded Systems Design

Snook C, Butler M (2006) UML-B : Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology (15):92–122

Tao L, Fengsheng J (2015) B Formal Modeling Based on UML Class. In: IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), 2015, IEEE Conference Publications, pp 1–6, dOI: 10.1109/ICSPCC.2015.7338854

Truong N, J S (2006) Verification of UML model elements using B. Information Science and Engineering (22):357–373

W3C (2014) XSL transformations (XSLT) version 3.0, W3C recommendation, oct. 2014 Available at: http://www.w3.org/TR/2014/WD-xslt-30-20141002/

Yang J (2009) A framework for formalizing UML models with formal language rCOS. In: Fourth International Conference on Frontier of Computer Science and Technology, 2009 (FCST'09), IEEE, pp 408–416

Appendix A
The FoCaLiZe specification generated from the UML/OCL model of Fig. 1

The complete FoCaLiZe specification generated by application of the proposed transformation rules on the UML/OCL model of Fig. 1 is presented as follows:

```
species Person =
 signature get_name : Self -> string;
 signature get_age : Self -> int;
 signature setAge:Self->int-> Self;
 signature birthdayHappens : Self -> Self;
 signature newPerson:string-> int-> Self;

(* mapping of OCL constraints on Person *)
 property inv_1: all p:Self,
                   (get_age(p) > 0);
 end;;

species FArray ( Obj is Setoid,
                 i in IntCollection) =
 signature get_data : Self -> list(Obj);
 signature isFull : Self -> bool ;
 signature isEmpty : Self -> bool ;
 signature length: Self -> int ;
 signature newInstance: list(Obj) -> Self;

(* mapping of OCL constraints on FArray *)
 property inv_1 : all s : Self,
            isEmpty(s) -> (length(s) = 0);
 property inv_2 : all s : Self,
             isEmpty(s) -> ~~(isFull(s));
 end;;

species FStack ( Obj is Setoid,
                 i in IntCollection ) =
 inherit FArray(Obj, i);
 signature head : Self -> Obj ;
 signature push : Obj -> Self -> Self;
 signature pop  :  Self -> Self;
(* mapping of OCL constraints on FStack *)
 property pre_post_push_1: all e: Obj,
                           all s: Self,
  isEmpty (s) -> isEmpty(pop(push(e, s)));

 property pre_post_push_2: all e:Obj,
                           all s:Self,
 ~~(isFull(s))-> equal(pop(push(e, s)), s);
 end;;

(* creation of the entity medeling
                 the integer value 100 *)
 let e = IntCollection!newInstance(100);;

species PersonStack( T is Person,
                     i in IntCollection)=
 inherit FStack(T, e);
 end;;
```

Appendix B
Refinements of the FoCaLiZe specification generated from the UML/OCL model of Fig. 1

A FoCaLiZe developer completes (refines) the abstract specification (generated from the UML/OCL model of Fig. 1) by providing definitions for all its undefined methods, as follows:

```
 (* The species Person_Def provides
    definitions for the species
                     Person methods *)
species Person_Def =  inherit Person;
 representation = string * (int) ;
 let newPerson(s :string, a:int ):Self =
                                  (s, a);
 let get_name(p:Self):string = fst(p);
 let get_age(p:Self):int = snd(p);
 let setAge (p:Self, a:int):Self =
              newPerson(get_name(p), a);
 let birthdayHappens (p:Self):Self =
  newPerson(get_name(p), (get_age(p)+ 1));

 (* The functions element and equal
          are inherited from Setoid *)
 let equal (x:Self, y:Self): bool =
             (get_name(x) = get_name(y));
 let element:Self = ("toto", 1);
 end;;
species PersonStack ( T is Person,
                      i in IntCollection)=
                   inherit FStack(T, e);
 representation = list(T);
 let newInstance(x: list(T))Self = x ;
 let get_data (x: Self):list(T)= x;
 let isFull(x:Self):bool = (length(x) =
             (IntCollection!to_int(i)));
 let isEmpty(x:Self):bool =
                        (length(x) = 0) ;
 let head(x: Self):T =
                   match get_data(x) with
  | [] ->
     focalize_error("The stack is empty")
  | y :: z -> y;

let push (o: T, y: Self):Self =
 if ~~(isFull(y))
 then newInstance(o::get_data(y))
 else focalize_error("The stack is full");

 let pop(x: Self):Self =
                 match get_data(x) with
  | [] ->
    focalize_error ("The stack is empty")
  | y :: z -> newInstance(z);
 let rec length(x: Self):int =
                   match (x) with
  | [] -> 0
  | y :: z -> 1 + length(z);
 (* The functions element and
        equal are inherited from Setoid *)
 let equal(x:Self, y:Self): bool = (x = y);
 let element:Self = [] ;
 end;;
```